Co-induction Simply
Automatic Co-inductive Proofs in a Program Verifier

K. Rustan M. Leino and Michał Moskal

Microsoft Research, Redmond, WA, USA
{leino,micmo}@microsoft.com

Abstract. This paper shows that an SMT-based program verifier can support reasoning about co-induction—handling infinite data structures, lazy function calls, and user-defined properties defined as greatest fix-points, as well as letting users write co-inductive proofs. Moreover, the support can be packaged to provide a simple user experience. The paper describes the features for co-induction in the language and verifier Dafny, defines their translation into input for a first-order SMT solver, and reports on some encouraging initial experience.

0 Introduction

Mathematical induction is a cornerstone of programming and program verification. It arises in data definitions (*e.g.*, some algebraic data structures can be described using induction [4]), it underlies program semantics (*e.g.*, it explains how to reason about finite iteration and recursion [1]), and it gets used in proofs (*e.g.*, supporting lemmas about data structures use inductive proofs [16]). Whereas induction deals with finite things (data, behavior, etc.), its dual, *co-induction*, deals with possibly infinite things. Co-induction, too, is important in programming and program verification, where it arises in data definitions (*e.g.*, lazy data structures [31]), semantics (*e.g.*, concurrency [29]), and proofs (*e.g.*, showing refinement in a co-inductive big-step semantics [22]). It is thus desirable to have good support for both induction and co-induction in a system for constructing and reasoning about programs.

Dramatic improvements in satisfiability-modulo-theories (SMT) solvers have brought about new levels of power in automated reasoning. Some program verifiers and interactive proof assistants have used this power to reduce the amount of human interaction needed to achieve results (*e.g.*, [12,8,17,5]). In this paper, we introduce the first SMT-based verifier to support co-induction.

The verifier is for programs written in the verification-aware programming language Dafny [17],[0] which we extend with co-inductive features. Co-datatypes and co-recursive functions make it possible to use lazily evaluated data structures (like in Haskell [31] or Agda [27]). Co-predicates, defined by greatest fix-points, let programs state properties of such data structures (as can also be done in, for example, Coq [3]). For the purpose of writing co-inductive proofs in the language, we introduce *co-lemmas*. Ostensibly, a co-lemma invokes the co-induction hypothesis much

[0] Dafny is an open-source project at `http://dafny.codeplex.com`.

```
// infinite streams
codatatype IStream⟨T⟩ = ICons(head: T, tail: IStream)
// pointwise product of streams
function Mult(a: IStream⟨int⟩, b: IStream⟨int⟩): IStream⟨int⟩
{ ICons(a.head * b.head, Mult(a.tail, b.tail)) }
// lexicographic order on streams
copredicate Below(a: IStream⟨int⟩, b: IStream⟨int⟩)
{ a.head ≤ b.head ∧ (a.head = b.head ⟹ Below(a.tail, b.tail)) }
// a stream a Below its Square
colemma Theorem_BelowSquare(a: IStream⟨int⟩)
  ensures Below(a, Mult(a, a));
{ assert a.head ≤ Mult(a, a).head;
  if a.head = Mult(a, a).head { Theorem_BelowSquare(a.tail); } }
// an incorrect property and a bogus proof attempt
colemma NotATheorem_SquareBelow(a: IStream⟨int⟩)
  ensures Below(Mult(a, a), a);  // ERROR
{ NotATheorem_SquareBelow(a); }
```

Fig. 0. A taste of how the co-inductive features in Dafny come together to give straightforward definitions of infinite matters. The proof of the theorem stated by the first co-lemma lends itself to the following intuitive reading: To prove that a is below Mult(a, a), check that their heads are ordered and, if the heads are equal, also prove that the tails are ordered. The second co-lemma states a property that does not always hold; the verifier is not fooled by the bogus proof attempt and instead reports the property as unproved. wJHo »[1]

like an inductive proof invokes the induction hypothesis. Underneath the hood, our co-inductive proofs are actually approached via induction [24]: co-lemmas provide a syntactic veneer around this approach. We are not aware of any other proof assistant with co-inductive constructs that takes this approach.

These language features and the automation in our SMT-based verifier combine to provide a simple view of co-induction. As a sneak peek, consider the program in Fig. 0.[1] It defines a type IStream of infinite streams, with constructor ICons and destructors head and tail. Function Mult performs pointwise multiplication on infinite streams of integers, defined using a co-recursive call (which is evaluated lazily). Co-predicate Below is defined as a greatest fix-point, which intuitively means that the co-predicate will take on the value **true** if the recursion goes on forever without determining a different value. The co-lemma states the theorem Below(a, Mult(a, a)). Its body gives the proof, where the recursive invocation of the co-lemma corresponds to an invocation of the co-induction hypothesis.

We argue that these definitions in Dafny are simple enough to level the playing field between induction (which is familiar) and co-induction (which, despite being the dual of induction, is often perceived as eerily mysterious). Moreover, the automation provided by our SMT-based verifier reduces the tedium in writing co-inductive proofs. For example, it verifies Theorem_BelowSquare from the program text given in Fig. 0—no additional lemmas or tactics are needed. (This is true throughout the paper—the

[1] The examples in the figures can be tried and tweaked online at the following address: http://rise4fun.com/Dafny/*id* where *id* (*e.g.*, wJHo) is provided below every figure.

verifier works from the given program text and does not require or accept any other input.) In fact, as a consequence of the automatic-induction heuristic in Dafny [18], the verifier will automatically verify `Theorem_BelowSquare` even given an empty body.

Just like there are restrictions on when an *inductive* hypothesis can be *invoked*, there are restriction on how a *co-inductive* hypothesis can be *used*. These are, of course, taken into consideration by our verifier. For example, as illustrated by the second co-lemma in Fig. 0, invoking the co-inductive hypothesis in an attempt to obtain the entire proof goal is futile. (We explain how this works in Sec. 2.1.)

Our initial experience with co-induction in Dafny shows it to provide an intuitive, low-overhead user experience that compares favorably to even the best of today's interactive proof assistants for co-induction. In addition, the co-inductive features and verification support in Dafny have other potential benefits. The features are a stepping stone for verifying functional lazy programs with Dafny. Co-inductive features have also shown to be useful in defining language semantics, as needed to verify the correctness of a compiler [22], so this opens the possibility that such verifications can benefit from SMT automation.

0.0 Contributions

- First SMT-based verifier for reasoning about co-induction.
- Language design that blends inductive and co-inductive features, allowing both recursive and *co-recursive calls* to the same function (Sec. 1).
- User-callable *prefix predicates*—finite unfoldings of co-predicates used to establish co-predicates via induction (Secs. 1.3 and 4).
- Extension of the technique of writing inductive proofs as programs (see [18]) to co-inductive proofs using *co-lemmas* (Sec. 2). Unlike tactic-based systems, these programs show the high-level structure of the (inductive and co-inductive) proofs. Yet the automation provided by the SMT solver makes it unnecessary to manually author the proof terms.
- Low-overhead tool-supported way to write and learn about co-inductive proofs (see examples in Sec. 3).

1 Co-inductive Definitions

In this section and the next, we describe the design of our co-inductive extension of Dafny. We start with the constructs for defining types, values, and properties of possibly infinite data structures. Though we will hint at how our design compares to the existing design for inductive constructs, space constraints prevent us from giving the details of those; to learn more, see [18,20].

1.0 Background

The Dafny programming language supports functions and methods. A *function* in Dafny is a mathematical function (*i.e.*, it is well-defined, deterministic, and pure), whereas a *method* is a body of statements that can mutate the state of the program. A function is defined by its given body, which is an expression. To ensure that function definitions are mathematically consistent, Dafny insists that recursive calls be well-founded, enforced as follows: Dafny computes the call graph of functions. The strongly connected components within it are *clusters* of mutually recursive definitions arranged in a DAG. This stratifies the functions so that a call from one cluster in the DAG to a lower cluster is allowed arbitrarily. For an intra-cluster call, Dafny prescribes a proof obligation that gets taken through the program verifier's reasoning engine. Semantically, each function activation is labeled by a *rank*—a lexicographic tuple determined by evaluating the function's **decreases** clause upon invocation of the function. The proof obligation for an intra-cluster call is thus that the rank of the callee is strictly less (in a language-defined well-founded relation) than the rank of the caller [17]. Because these well-founded checks correspond to proving termination of executable code, we will often refer to them as "termination checks". The same process applies to methods.

Lemmas in Dafny are commonly introduced by declaring a method, stating the property of the lemma in the *postcondition* (keyword **ensures**) of the method, perhaps restricting the domain of the lemma by also giving a *precondition* (keyword **requires**), and using the lemma by invoking the method [14,18]. Lemmas are stated, used, and proved as methods, but since they have no use at run time, such lemma methods are typically declared as *ghost*, meaning that they are not compiled into code. The keyword **lemma** introduces such a method. Control flow statements correspond to proof techniques—case splits are introduced with **if** statements, recursion and loops are used for induction, and method calls for structuring the proof. Additionally, the statement:

```
forall x | P(x) { Lemma(x); }
```

is used to invoke `Lemma(x)` on all `x` for which `P(x)` holds. If `Lemma` ensures `Q(x)`, then the **forall** statement establishes $\forall$ `x • P(x)` $\Longrightarrow$ `Q(x)`.

1.1 Defining Co-inductive Datatypes

Each value of an *inductive datatype* is finite, in the sense that it can be constructed by a finite number of calls to datatype constructors. In contrast, values of a *co-inductive datatype*, or *co-datatype* for short, can be infinite. For example, a co-datatype can be used to represent infinite trees.

Syntactically, the declaration of a co-datatype in Dafny looks like that of a datatype, giving prominence to the constructors (following Coq [10]). For example, Fig. 1 defines a co-datatype `Stream` of possibly infinite lists. Analogous to the common finite list datatype, `Stream` declares two constructors, `SNil` and `SCons`. Values can be destructed using **match** expressions and statements. In addition, like for inductive datatypes, each constructor `C` automatically gives rise to a discriminator `C?` and each parameter of a constructor can be named in order to introduce a corresponding destructor. For example, if xs is the stream `SCons(`x, ys`)`, then xs`.SCons?` and xs`.head` $=x$ hold. In contrast to datatype declarations, there is no grounding check for co-datatypes—since a co-datatype admits infinite values, the type is nevertheless inhabited.

```
codatatype Stream⟨T⟩ = SNil | SCons(head: T, tail: Stream)
function Up(n: int): Stream⟨int⟩ { SCons(n, Up(n+1)) }
function FivesUp(n: int): Stream⟨int⟩
  decreases 4 - (n - 1) % 5;
{ if n % 5 = 0 then SCons(n, FivesUp(n+1)) else FivesUp(n+1) }
```

Fig. 1. `Stream` is a co-inductive datatype whose values are possibly infinite lists. Function `Up` returns a stream consisting of all integers upwards of n and `FivesUp` returns a stream consisting of all multiples of 5 upwards of n. The self-call in `Up` and the first self-call in `FivesUp` sit in productive positions and are therefore classified as co-recursive calls, exempt from termination checks. The second self-call in `FivesUp` is not in a productive position and is therefore subject to termination checking; in particular, each recursive call must decrease the rank defined by the **decreases** clause. `CplhV` »

1.2 Creating Values of Co-datatypes

To define values of co-datatypes, one could imagine a "co-function" language feature: the body of a "co-function" could include possibly never-ending self-calls that are interpreted by a greatest fix-point semantics (akin to a `CoFixpoint` in Coq). Dafny uses a different design: it offers only functions (not "co-functions"), but it classifies each intra-cluster *call* as either *recursive* or *co-recursive*. Recursive calls are subject to termination checks [17]. Co-recursive calls may be never-ending, which is what is needed to define infinite values of a co-datatype. For example, function `Up(`n`)` in Fig. 1 is defined as the stream of numbers from n upward: it returns a stream that starts with n and continues as the co-recursive call `Up(`$n+1$`)`.

To ensure that co-recursive calls give rise to mathematically consistent definitions, they must occur only in *productive positions*. This says that it must be possible to determine each successive piece of a co-datatype value after a finite amount of work. This condition is satisfied if every co-recursive call is syntactically *guarded* by a constructor of a co-datatype, which is the criterion Dafny uses to classify intra-cluster calls as being either co-recursive or recursive. Calls that are classified as co-recursive are exempt from termination checks.

A consequence of the productivity checks and termination checks is that, even in the absence of talking about least or greatest fix-points of self-calling functions, all functions in Dafny are deterministic. Since there is no issue of several possible fix-points, the language allows one function to be involved in both recursive and co-recursive calls, as we illustrate by the function `FivesUp` in Fig. 1.

1.3 Stating Properties of Co-datatypes

Determining properties of co-datatype values may require an infinite number of observations. To that avail, Dafny provides *co-predicates*. Self-calls to a co-predicate need not terminate. Instead, the value defined is the greatest fix-point of the given recurrence equations. Figure 2 defines a co-predicate that holds for exactly those streams whose payload consists solely of positive integers.

Some restrictions apply. To guarantee that the greatest fix-point always exists, the (implicit functor defining the) co-predicate must be monotonic. This is enforced by

```
copredicate Pos(s: Stream⟨int⟩)
{ match s
  case SNil ⇒ true
  case SCons(x, rest) ⇒ x > 0 ∧ Pos(rest) }
// Automatically generated by the Dafny compiler:
predicate Pos#[_k: nat](s: Stream⟨int⟩)
  decreases _k;
{ if _k = 0 then true else
    match s
    case SNil ⇒ true
    case SCons(x, rest) ⇒ x > 0 ∧ Pos#[_k-1](rest) }
```

Fig. 2. A co-predicate `Pos` that holds for those integer streams whose every integer is greater than 0. The co-predicate definition implicitly also gives rise to a corresponding prefix predicate, `Pos`$^{\#}$. The syntax for calling a prefix predicate sets apart the argument that specifies the prefix length, as shown in the last line; for this figure, we took the liberty of making up a coordinating syntax for the signature of the automatically generated prefix predicate. `eYml` »

a syntactic restriction on the form of the body of co-predicates: after conversion to negation normal form (*i.e.*, pushing negations down to the atoms), intra-cluster calls of co-predicates must appear only in *positive* positions—that is, they must appear as atoms and must not be negated. Additionally, to guarantee soundness later on, we require that they appear in *co-friendly* positions—that is, in negation normal form, when they appear under existential quantification, the quantification needs to be limited to a finite range.[2] Since the evaluation of a co-predicate might not terminate, co-predicates are always ghost. There is also a restriction on the call graph that a cluster containing a co-predicate must contain only co-predicates, no other kinds of functions.

A **`copredicate`** declaration of `P` defines not just a co-predicate, but also a corresponding *prefix predicate* `P`$^{\#}$. A prefix predicate is a finite unrolling of a co-predicate. The prefix predicate is constructed from the co-predicate by

– adding a parameter _k of type **`nat`** to denote the *prefix length*,
– adding the clause **`decreases`** `_k;` to the prefix predicate (the co-predicate itself is not allowed to have a **`decreases`** clause),
– replacing in the body of the co-predicate every intra-cluster call `Q(` *args* `)` to a co-predicate by a call `Q`$^{\#}$`[_k` $-$ `1](` *args* `)` to the corresponding prefix predicate, and then
– prepending the body with **`if`** `_k = 0` **`then true else`**.

For example, for co-predicate `Pos`, the definition of the prefix predicate `Pos`$^{\#}$ is as suggested in Fig. 2. Syntactically, the prefix-length argument passed to a prefix predicate to indicate how many times to unroll the definition is written in square brackets, as in `Pos`$^{\#}$`[` k `](` s `)`. The definition of `Pos`$^{\#}$ is available only at clusters strictly higher than that of `Pos`; that is, `Pos` and `Pos`$^{\#}$ must not be in the same cluster. In other words, the definition of `Pos` cannot depend on `Pos`$^{\#}$.

[2] Higher-order function support in Dafny is rather modest and typical reasoning patterns do not involve them, so this restriction is not as limiting as it would have been in, *e.g.*, Coq.

```
lemma UpPosLemma(n: int)
  requires n > 0;
  ensures Pos(Up(n));
{ forall k | 0 ≤ k { UpPosLemmaK(k, n); } }
lemma UpPosLemmaK(k: nat, n: int)
  requires n > 0;
  ensures Pos#[k](Up(n));
  decreases k;
{ if k ≠ 0 {
    // this establishes Pos#[k-1](Up(n).tail)
    UpPosLemmaK(k-1, n+1); } }
```

Fig. 3. The lemma `UpPosLemma` proves `Pos(Up(`n`))` for every $n > 0$. We first show `Pos`$^{\#}$`[`k`](Up(`n`))`, for $n > 0$ and an arbitrary k, and then use the **`forall`** statement to show $\forall$ `k` $\bullet$ `Pos`$^{\#}$`[k](Up(`n`))`. Finally, the axiom $\mathcal{D}$`(Pos)` is used (automatically) to establish the co-predicate. `d7J3` »

Equality between two values of a co-datatype is a built-in co-predicate. It has the usual equality syntax $s = t$, and the corresponding prefix equality is written $s =^{\#}$`[`k`]` t.

2 Co-inductive Proofs

From what we have said so far, a program can make use of properties of co-datatypes. For example, a method that declares `Pos(`s`)` as a precondition can rely on the stream s containing only positive integers. In this section, we consider how such properties are established in the first place.

2.0 Properties About Prefix Predicates

Among other possible strategies for establishing co-inductive properties (*e.g.*, [13,7,**?**]), we take the time-honored approach of reducing co-induction to induction [24]. More precisely, Dafny passes to the SMT solver an assumption $\mathcal{D}(P)$ for every co-predicate P, where:

$$\mathcal{D}(P) \quad \equiv \quad \forall\, \overline{x} \bullet\ P(\overline{x}) \iff \forall\, k \bullet\ P^{\#k}(\overline{x})$$

In Sec. 4, we state a soundness theorem of such assumptions, provided the co-predicates meet the co-friendly restrictions from Sec. 1.3. An example proof of `Pos(Up(`n`))` for every $n > 0$ is shown in Fig. 3.

2.1 Co-lemmas

As we just showed, with help of the $\mathcal{D}$ axiom we can now prove a co-predicate by inductively proving that the corresponding prefix predicate holds for all prefix lengths k. In this section, we introduce *co-lemma* declarations, which bring about two benefits. The first benefit is that co-lemmas are syntactic sugar and reduce the tedium of having to write explicit quantifications over k. The second benefit is that, in simple cases, the bodies of co-lemmas can be understood as co-inductive proofs directly. As an example,

```
colemma UpPosLemma(n: int)
  requires n > 0;
  ensures Pos(Up(n));
{ UpPosLemma(n+1); }
```

Fig. 4. A proof of the lemma from Fig. 3 using the syntactic sugar of a co-lemma. Among other things, the call to `UpPosLemma(n+1)` is desugared to `UpPosLemma#[_k-1](n+1)` (which can also be used directly) and the proof goal is desugared to `Pos#[_k](Up(n))`. Intuitively, the body of the co-lemma simply invokes the co-induction hypothesis to complete the proof. `Se7h` »

consider the co-lemma in Fig. 4, which can be understood as follows: `UpPosLemma` invokes itself co-recursively to obtain the proof for `Pos(Up(n).tail)` (since `Up(n).tail` equals `Up(n+1)`). The proof glue needed to then conclude `Pos(Up(n))` is provided automatically, thanks to the power of the SMT-based verifier.

2.2 Prefix Lemmas

To understand why the code in Fig. 4 is a sound proof, let us now describe the details of the desugaring of co-lemmas. In analogy to how a **`copredicate`** declaration defines both a co-predicate and a prefix predicate, a **`colemma`** declaration defines both a co-lemma and *prefix lemma*. In the call graph, the cluster containing a co-lemma must contain only co-lemmas and prefix lemmas, no other methods or function. By decree, a co-lemma and its corresponding prefix lemma are always placed in the same cluster. Both co-lemmas and prefix lemmas are always ghosts.

The prefix lemma is constructed from the co-lemma by

- adding a parameter _k of type **`nat`** to denote the prefix length,
- replacing in the co-lemma's postcondition the positive co-friendly occurrences of co-predicates by corresponding prefix predicates, passing in _k as the prefix-length argument,
- prepending _k to the (typically implicit) **`decreases`** clause of the co-lemma,
- replacing in the body of the co-lemma every intra-cluster call `M(` *args* `)` to a co-lemma by a call `M#[_k` $-$ `1](` *args* `)` to the corresponding prefix lemma, and then
- making the body's execution conditional on $\_k \neq 0$.

Note that this rewriting removes all co-recursive calls of co-lemmas, replacing them with recursive calls to prefix lemmas. These recursive call are, as usual, checked to be terminating. We allow the pre-declared identifier _k to appear in the original body of the co-lemma.[3]

We can now think of the body of the co-lemma as being replaced by a **`forall`** call, for every k, to the prefix lemma. By construction, this new body will establish the co-lemma's declared postcondition (on account of the $\mathcal{D}$ axiom, which we prove sound in Sec. 4, and remembering that only the positive co-friendly occurrences of co-predicates in the co-lemma's postcondition are rewritten), so there is no reason for the program verifier to check it.

[3] Note, two places where co-predicates and co-lemmas are not analogous are: co-predicates must not make recursive calls to their prefix predicates, and co-predicates cannot mention _k.

The actual desugaring of Fig. 4 is in fact the code from Fig. 3, except that `UpPosLemmaK` is named `UpPosLemma`$^{\#}$ and modulo a minor syntactic difference in how the k argument is passed.

In the recursive call of the prefix lemma, there is a proof obligation that the prefix-length argument $\_k - 1$ is a natural number. Conveniently, this follows from the fact that the body has been wrapped in an **if** $\_k \neq 0$ statement. This also means that the postcondition must hold trivially when $\_k = 0$, or else a postcondition violation will be reported. This is an appropriate design for our desugaring, because co-lemmas are expected to be used to establish co-predicates, whose corresponding prefix predicates hold trivially when $\_k = 0$. (To prove other predicates, use an ordinary lemma, not a co-lemma.)

It is interesting to compare the intuitive understanding of the co-inductive proof in Fig. 4 with the inductive proof in Fig. 3. Whereas the inductive proof is performing proofs for deeper and deeper equalities, the co-lemma can be understood as producing the infinite proof on demand.

2.3 Automation

Because co-lemmas are desugared into lemmas whose postconditions benefit from induction, Dafny's usual induction tactic kicks in [18]. Effectively, it adds a **forall** statement at the beginning of the prefix lemma's body, invoking the prefix lemma recursively on all smaller tuples of arguments. Typically, the useful argument tuples are those with a smaller value of the implicit parameter _k and any other values for the other parameters, but the **forall** statement will also cover tuples with the same _k and smaller values of the explicit parameters.

Thanks to the induction tactic, the inductive lemma `UpPosLemmaK` from Fig. 3 is verified automatically even if it is given an empty body. So, co-lemma `UpPosLemma` in Fig. 4 is also verified automatically even if given an empty body—it is as if Dafny had a tactic for automatic co-induction as well.

3 More Examples

In this section, we further illustrative what can easily be achieved with our co-induction support in Dafny. We use examples that other treatments of co-induction have used or offered as challenges. We give links to these examples online (cf. Footnote 1), but also point out that most of the examples are also available in the Dafny test suite (see Footnote 0).

Zip Figure 5 states a few properties of the `zip` function on infinite streams. (See the figure caption for a more detailed description.)

Wide Trees Figure 6 shows a definition of trees with infinite width but finite height.

```
codatatype IStream⟨T⟩ = ICons(head: T, tail: IStream)
function zip(xs: IStream, ys: IStream): IStream
{ ICons(xs.head, ICons(ys.head, zip(xs.tail, ys.tail))) }
function even(xs: IStream): IStream { ICons(xs.head, even(xs.tail.tail)) }
function odd(xs: IStream): IStream { even(xs.tail) }
function bzip(xs: IStream, ys: IStream, f: bool) : IStream
{ if f then ICons(xs.head, bzip(xs.tail, ys, ¬f))
  else ICons(ys.head, bzip(xs, ys.tail, ¬f)) }
colemma EvenOddLemma(xs: IStream)
  ensures zip(even(xs), odd(xs)) = xs;
    { EvenOddLemma(xs.tail.tail); }
colemma EvenZipLemma(xs: IStream, ys: IStream)
  ensures even(zip(xs, ys)) = xs;
    { /* Automatic. */ }
colemma BzipZipLemma(xs: IStream, ys: IStream)
  ensures zip(xs, ys) = bzip(xs, ys, true);
    { BzipZipLemma(xs.tail, ys.tail); }
```

Fig. 5. Some standard examples of combining and dividing infinite streams (cf. [11]). The proof of EvenZipLemma is fully automatic, whereas the others require a single recursive call to be made explicitly. The **forall** statement inserted automatically by Dafny's induction tactic is in principle strong enough to prove each of the three lemmas, but the incompleteness of reasoning with quantifiers in SMT solvers makes the explicit calls necessary. wq7Y »

FivesUp The function FivesUp defined in Fig. 1 calls itself both recursively and co-recursively. To prove that FivesUp(n) satisfies Pos for any positive n requires the use of induction and co-induction together (which may seem mind boggling). We give a simple proof in Fig. 7.

Recall that the **decreases** clause of the prefix lemma implicitly starts with _k, so the termination check for each of the recursive calls passes: the first call decreases _k, whereas the second call decreases the expression given explicitly. We were delighted to see that the **decreases** clause (copied from the definition of FivesUp) is enough of a hint to Dafny; it needs to be supplied manually, but the body of the co-lemma can in fact be left empty.

Filter The central issue in the FivesUp example is also found in the more useful *filter* function. It has a straightforward definition in Dafny:

```
function Filter(s: IStream): IStream
  requires AlwaysAnother(s);
  decreases Next(s);
{ if P(s.head) then ICons(s.head, Filter(s.tail)) else Filter(s.tail) }
```

In the **else** branch, Filter calls itself recursively. The difficulty is proving that this recursion terminates. In fact, the recursive call would not terminate given an arbitrary stream; therefore, Filter has a precondition that elements satisfying P occur infinitely often. To show progress toward the subsequent element of output, function Next counts the number of steps in the input s until the next element satisfying P.

```
datatype Tree = Node(children: Stream⟨Tree⟩)
predicate IsFiniteHeight(t: Tree) { ∃ n • 0 ≤ n ∧ LowerThan(t.children, n) }
copredicate LowerThan(s: Stream⟨Tree⟩, n: nat)
{ match s
  case SNil ⇒ true
  case SCons(t, tail) ⇒
    1 ≤ n ∧ LowerThan(t.children, n-1) ∧ LowerThan(tail, n) }
```

Fig. 6. By itself, the datatype declaration `Tree` will allow structures that are infinite in height (the situation in Agda is similar [0]). In Dafny, the part of a `Tree` that can be inducted over is finite, in fact of size just 1 (for more details of such induction, see [20]). To describe trees that are possibly infinite only in width (that is, with finite height, but each node having a possibly infinite number of children), we declare a predicate `IsFiniteHeight`. The use of a predicate to characterize an interesting subset of a type is typical in Dafny (also in the imperative parts of the language; for example, class invariants are just ordinary predicates [17]). `nU5e` »

```
colemma FivesUpPos(n: int)
  requires n > 0;
  ensures Pos(FivesUp(n));
  decreases 4 - (n - 1) % 5;
{ if n % 5 = 0 { FivesUpPos#[_k-1](n + 1); }
  else { FivesUpPos#[_k](n + 1); } }
```

Fig. 7. A proof that, for any positive n, all values in the stream `FivesUp(`n`)` are positive. The proof uses both induction and co-induction. To illustrate what is possible, we show both calls as explicitly targeting the prefix lemma. Alternatively, the first call could have been written as a call `FivesUpPos(`$n+1$`)` to the co-lemma, which would desugar to the same thing and would more strongly suggest the intuition of appealing to the co-inductive hypothesis. `7hNCq` »

The full example [19] defines the auxiliary functions and proves some theorems about `Filter`, see `8oeR` ». The filter function has also been formalized (with more effort) in other proof assistants, for example by Bertot in Coq [2].

Iterates In a paper that shows co-induction being encoded in the proof assistant Isabelle/HOL, Paulson [30] defines a function `Iterates(f, M)` that returns the stream

$$M,\ f(M),\ f^2(M),\ f^3(M),\ \ldots$$

In Dafny syntax, the function is defined as

```
function Iterates⟨A⟩(M: A): Stream⟨A⟩ { SCons(M, Iterates(f(M))) }
```

Paulson defines a function `Lmap`:

```
function Lmap(s: Stream): Stream
{ match s
  case SNil ⇒ SNil
  case SCons(a, tail) ⇒ SCons(f(a), Lmap(tail)) }
```

and proves that any function `h` satisfying `h(M) = SCons(M, Lmap(h(M)))` is indeed the function `Iterates`. This proof and all other examples from Paulson's paper can be done in Dafny, see `iplnx` ».

```
codatatype RecType = Bottom | Top | Arrow(dom: RecType, ran: RecType)
copredicate Subtype(a: RecType, b: RecType)
{
  a = Bottom ∨
  b = Top ∨
  (a.Arrow? ∧ b.Arrow? ∧ Subtype(b.dom, a.dom) ∧ Subtype(a.ran, b.ran))
}
colemma Subtype_Is_Transitive(a: RecType, b: RecType, c: RecType)
  requires Subtype(a, b) ∧ Subtype(b, c);
  ensures Subtype(a, c);
{
  if a ≠ Bottom ∧ c ≠ Top {
    Subtype_Is_Transitive(c.dom, b.dom, a.dom);
    Subtype_Is_Transitive(a.ran, b.ran, c.ran);
  }
}
```

Fig. 8. A definition of subtyping among recursive types. The co-lemma proves the subtype relation to be transitive.

Recursive Types Kozen and Silva also argue that the playing field between induction and co-induction can be leveled [**?**]. We have encoded all their examples in Dafny, see `yqel` », and show one of them in Fig. 8.

Big-step semantics Leroy [21] defines a co-inductive big-step semantics for the λ-calculus as follows:

$$\frac{}{\lambda x.m \overset{co}{\Rightarrow} \lambda x.m}\ \text{(id)} \qquad \frac{m_0 \overset{co}{\Rightarrow} \lambda x.m' \quad m_1 \overset{co}{\Rightarrow} n' \quad m'[x := n'] \overset{co}{\Rightarrow} n}{m_0 m_1 \overset{co}{\Rightarrow} n}\ \text{(beta)}$$

The double lines indicate that the proof tree is allowed to be infinite, with a greatest fix-point semantics. The intention is that if evaluation of m does not terminate, then $\forall\, n \bullet\ m \overset{co}{\Rightarrow} n$. Figure 9 gives the corresponding definition in Dafny.

4 Soundness

In this section, we formalize and prove the connection between co-predicates and prefix predicates. More precisely, we state a theorem that $\forall\, k \bullet\ P^{\#k}(\overline{x})$ is the greatest fix-point solution of the equation defining $P(\overline{x})$.

Consider a given cluster of co-predicate definitions, that is, a strongly connected component of co-predicates:

$$P_i(\overline{x_i}) = C_i \quad \text{for } i = 0 \ldots n \tag{0}$$

The right-hand sides (C_i) can reference functions, co-predicates, and prefix predicates from lower clusters, as well as co-predicates (P_j) in the same cluster. According to our restrictions in Sec. 1.3, the cluster contains only co-predicates, no prefix predicates or

```
datatype Term = Var(idx: nat) | Fun(Term) | App(m0: Term, m1: Term)
codatatype PreProof = Id | Beta(m: Term, n: Term, a: PreProof, b: PreProof, c: PreProof)
copredicate IsProof(m: Term, n: Term, d: PreProof)
{ match d
  case Id ⇒ m.Fun? ∧ m = n
  case Beta(m', n', a, b, c) ⇒ m.App? ∧ IsProof(m.m0, Fun(m'), a) ∧
          IsProof(m.m1, n', b) ∧ IsProof(Subst(m', 0, n'), n, c) }
predicate Eval(m: Term, n: Term) { ∃ d • IsProof(m, n, d) }
// ERROR - Eval' used in non-co-friendly position
copredicate Eval'(m: Term, n: Term)
{ match m
  case Fun(_) ⇒ m = n
  case App(m0, m1) ⇒ ∃ m', n' •
    Eval'(m0, Fun(m')) ∧ Eval'(m1, n') ∧ Eval'(Subst(m', 0, n'), n) }
```

Fig. 9. Big-step semantics definition in Dafny using De Bruijn indices. Explicit proof trees let the user provide witnesses to the SMT solver and work around the co-friendliness restriction. The alternative `Eval'` definition above does not pass the co-friendliness test, as it quantifies over `m'` and `n'` in every step. Full example and proof of $(\lambda x.\, x\, x)(\lambda x.\, x\, x) \overset{co}{\Rightarrow} m$ for all m can be found at uKXM »

other functions; so, any prefix predicate referenced in C_i is necessarily from a lower cluster.

A cluster can be syntactically reduced to a single co-predicate, *e.g.*:

$$\begin{array}{l} P(i, \overline{x_0}, \ldots, \overline{x_n}) = 0 \le i \le n \wedge ((i = 0 \wedge C_0\sigma) \vee \ldots \vee (i = n \wedge C_n\sigma)) \\ \text{where } \sigma \;=\; [P_i := (\lambda\, \overline{x_i} \bullet\; P(i, \overline{x_0}, \ldots, \overline{x_n}))\,]_{i=0}^{n} \end{array} \tag{1}$$

In what follows, we assume $P(x) = C_x$ to be the definition of P, where x stands for the tuple of arguments and C_x for the body above. Let:

$$F(A) = \{x \mid C_x[P := A]\} \tag{2}$$

where $C_x[P := A]$ is C_x with occurrences of P replaced with (the characteristic function of set) A. In other words, F is the functor taking an interpretation A of P and returning a new interpretation. In Sec. 1.3, we defined the semantics of a co-predicate to be the greatest fix-point of F (*i.e.*, $gfp(F)$).

Let $P^{\#}$ be the prefix predicate corresponding to P. We will write the prefix-length argument k as a superscript, as in $P^{\#k}$. The prefix predicates are defined inductively as follows:

$$P^{\#0}(x) \;\equiv\; \top \qquad\qquad P^{\#k+1}(x) \;\equiv\; C_x[P := P^{\#k}] \tag{3}$$

Theorem 0.

$$x \in gfp(F) \iff \forall\, k \bullet\; P^{\#k}(x)$$

The simple proof, which is found in our companion technical report [20], uses the Kleene fix-point theorem and the fact that F is Scott continuous (*i.e.*, intuitively, monotonic due to positivity restrictions, and possible to falsify with a finite number of argument tuples due to co-friendliness).

5 Related Work

Most previous attempts at verifying properties of programs using co-induction have been limited to program verification environments embedded in interactive proof assistants. Early work includes an Isabelle/HOL package for reasoning about fix-points and applying them to inductive and co-inductive definitions [30]. The package was building from first principles and apparently lacked much automation. Later, a variant of the *circular co-induction* proof rules [32] was used in the CoCasl [11] object-oriented specification system in Isabelle/HOL. These rules essentially give a way to hide away the co-induction hypothesis when it is first introduced, “freezing” it until a time when it is sound to use it. In CoCasl, as in the CIRC [23] prover embedded in the Maude term rewriting system, the automation is quite good. However, the focus is on proving equalities of co-datatype values, and expressing general co-predicates is not as direct as it is in Dafny.

Co-induction has long history in the Coq interactive proof assistant [10,7]. A virtue of the standard co-induction tactic in Coq is that the entire proof goal becomes available as the co-induction hypothesis. One must then discipline oneself to avoid using it except in productive instances, something that is not checked until the final `Qed` command. In Dafny, any **`assert`** in the middle of the proof will point out non-productive uses.

The language and proof assistant Agda [27,6], which uses dependent types based on intuitionistic type theory, has some support for co-induction. Co-recursive datatypes and calls are indicated in the program text using the operators ∞ and $\sharp$ (see, *e.g.*, [0]). In Agda, proof terms are authored manually; there is no tactic language and no SMT support to help with automation.

Using the *sized types* in MiniAgda [**?**], one also proves properties of infinite structures by proving them for any finite unrolling. Properties are specified using definitions of co-datatypes, which are more restrictive than co-predicates in Dafny. In particular, there is no existential quantification and thus co-friendliness comes for free.

Moore has verified the correctness of a compiler for the small language Piton [25]. The correctness theorem considers a run of k steps of a Piton program and shows that m steps of the compiled version of the program behave like the original, where m is computed as a function of k and k is an arbitrary natural number. One might also be interested in proving the compiler correctness for infinite runs of the Piton program, which could perhaps be facilitated by defining the Piton semantics co-inductively (*cf.* [21]). If the semantics-defining co-predicates satisfied our co-friendly restriction, then our $\mathcal{D}$ axiom would reduce reasoning about infinite runs to reasoning about all finite prefixes of those runs.

Our technique of handling co-induction can be applied in any prover that readily handles induction. This includes verifiers like VCC [8] and VeriFast [12], but also interactive proof assistants. As shown in Fig. 7, induction and co-induction can benefit from the same automation techniques, so we consider this line of inquiry promising.

6 Conclusions

We have presented a technique for reasoning about co-inductive properties, which requires only minor extensions of a verifier that already supports induction. In Dafny, the induction itself is built on top of off-the-shelf state-of-the-art first-order SMT technology [9], which provides high automation. In our initial experience, the co-inductive definitions and proofs seem accessible to users without a large degree of clutter. The striking similarity of the inductive and co-inductive proofs certainly helps here. Even so, we suspect that further automation is possible once techniques for mechanized co-induction reach a maturity more akin to what is provided for induction by tools today (*e.g.*, [16,28,26,3,15,33,18]). With possible applications in both verifiers and other proof assistants, our work of making co-induction available in an SMT-based verifier takes a step in the direction of reducing the human effort required to reason about co-induction.

Acknowledgments During the course of this work, we have benefited from discussions with many colleagues who understand co-induction far better than we. We are grateful to all and mention here a subset: Jasmin Blanchette, Manfred Broy, Adam Chlipala, Ernie Cohen, Patrick Cousot, Jean-Christophe Filliâtre, Bart Jacobs (Nijmegen), Daan Leijen, Ross Tate. We also thank many anonymous reviewers.

References

0. T. Altenkirch and N. A. Danielsson. Termination checking in the presence of nested inductive and coinductive types. Short note supporting a talk given at PAR 2010, 2010. Available from `http://www.cse.chalmers.se/~nad/publications/`.
1. B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
2. Y. Bertot. Filters on coinductive streams, an application to Eratosthenes' sieve. In *TLCA 2005*, volume 3461 of *LNCS*, pages 102–115. Springer, 2005.
3. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
4. R. Bird and P. Wadler. *Introduction to Functional Programming*. International Series in Computing Science. Prentice Hall, 1992.
5. S. Böhme and T. Nipkow. Sledgehammer: Judgement Day. In *IJCAR 2010*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010.
6. A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda — a functional language with dependent types. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.
7. A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, To appear. http://adam.chlipala.net/cpdt/.
8. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
9. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
10. E. Giménez. An application of co-inductive types in Coq: Verification of the alternating bit protocol. In *TYPES'95*, volume 1158 of *LNCS*, pages 135–152. Springer, 1996.
11. D. Hausmann, T. Mossakowski, and L. Schröder. Iterative circular coinduction for CoCasl in Isabelle/HOL. In *FASE*, volume 3442 of *LNCS*, pages 341–356. Springer, 2005.
12. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.
13. B. Jacobs and J. Rutten. An introduction to (co)algebra and (co)induction. In *Advanced Topics in Bisimulation and Coinduction*, number 52 in Cambridge Tracts in Theoretical Computer Science, pages 38–99. Cambridge University Press, 2011.

14. B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative programs as proofs. In *VS-Tools workshop at VSTTE 2010*, 2010.
15. M. Johansson, L. Dixon, and A. Bundy. Case-analysis for Rippling and inductive proof. In *ITP 2010*, volume 6172 of *LNCS*, pages 291–306. Springer, 2010.
16. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
17. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
18. K. R. M. Leino. Automating induction with an SMT solver. In *VMCAI 2012*, volume 7148 of *LNCS*, pages 315–331. Springer, 2012.
19. K. R. M. Leino. Automating theorem proving with SMT. In *ITP 2013*, volume 7998 of *LNCS*, pages 2–16. Springer, 2013.
20. K. R. M. Leino and M. Moskal. Co-induction simply: Automatic co-inductive proofs in a program verifier. Technical Report MSR-TR-2013-49, Microsoft Research, 2013.
21. X. Leroy. Coinductive big-step operational semantics. In *ESOP 2006*, volume 3924 of *LNCS*, pages 54–68. Springer, 2006.
22. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
23. D. Lucanu and G. Rosu. CIRC: A circular coinductive prover. In *CALCO*, volume 4624 of *LNCS*, pages 372–378. Springer, 2007.
24. R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.
25. J. S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.
26. T. Nipkow, L. Paulson, and M. Menzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
27. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
28. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV '96*, volume 1102 of *LNCS*, pages 411–414. Springer, 1996.
29. D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conference*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.
30. L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7, 1997.
31. S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.
32. G. Rosu and D. Lucanu. Circular coinduction: A proof theoretical foundation. In *CALCO*, volume 5728 of *LNCS*, pages 127–144. Springer, 2009.
33. W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *TACAS*, volume 7214 of *LNCS*, pages 407–421. Springer, 2012.